Motivating Examples

- Simulating Ocean Currents
  - Regular structure, scientific computing
- Simulating the Evolution of Galaxies
  - Irregular structure, scientific computing
- Rendering Scenes by Ray Tracing
  - Irregular structure, computer graphics
  - Not discussed here (read in book)

Simulating Ocean Currents

- Model as two-dimensional grids
- Discretize in space and time
  - finer spatial and temporal resolution => greater accuracy
- Many different computations per time step
  - set up and solve equations
- Concurrency across and within grid computations

Simulating Galaxy Evolution

- Simulate the interactions of many stars evolving over time
- Computing forces is expensive
- \( O(n^2) \) brute force approach
- Hierarchical Methods take advantage of force law: \( \frac{G m_1 m_2}{r^2} \)
  - Many time-steps, plenty of concurrency across stars within one
Rendering Scenes by Ray Tracing

- Shoot rays into scene through pixels in image plane
- Follow their paths
  - they bounce around as they strike objects
  - they generate new rays: ray tree per input ray
- Result is color and opacity for that pixel
- Parallelism across rays

All case studies have abundant concurrency

Parallel Programming Task

Break up computation into tasks
- assign tasks to processors

Break up data into chunks
- assign chunks to memories

Introduce synchronization for:
- mutual exclusion
- event ordering

Steps in Creating a Parallel Program

4 steps: Decomposition, Assignment, Orchestration, Mapping
- Done by programmer or system software (compiler, runtime, ...)
- Issues are the same, so assume programmer does it all explicitly

Partitioning for Performance

Balancing the workload and reducing wait time at synch points
Reducing inherent communication
Reducing extra work

Even these algorithmic issues trade off:
- Minimize comm. => run on 1 processor => extreme load imbalance
- Maximize load balance => random assignment of tiny tasks => no control over communication
- Good partition may imply extra work to compute or manage it

Goal is to compromise
- Fortunately, often not difficult in practice
Load Balance and Synch Wait Time

Limit on speedup: \( \text{Speedup}_{\text{problem}}(p) \leq \frac{\text{Sequential Work}}{\text{Max Work on any Processor}} \)
- Work includes data access and other costs
- Not just equal work, but must be busy at same time

Four parts to load balance and reducing synch wait time:
1. Identify enough concurrency
2. Decide how to manage it
3. Determine the granularity at which to exploit it
4. Reduce serialization and cost of synchronization

Deciding How to Manage Concurrency

Static versus Dynamic techniques

Static:
- Algorithmic assignment based on input; won't change
- Low runtime overhead
- Computation must be predictable
- Preferable when applicable (except in multiprogrammed/heterogeneous environment)

Dynamic:
- Adapt at runtime to balance load
- Can increase communication and reduce locality
- Can increase task management overheads

Dynamic Assignment

Profile-based (semi-static):
- Profile work distribution at runtime, and repartition dynamically
- Applicable in many computations, e.g. Barnes-Hut, some graphics

Dynamic Tasking:
- Deal with unpredictability in program or environment (e.g. Raytrace)
  - computation, communication, and memory system interactions
  - multiprogramming and heterogeneity
  - used by runtime systems and OS too
- Pool of tasks: take and add tasks until done
- E.g. “self-scheduling” of loop iterations (shared loop counter)

Dynamic Tasking with Task Queues

Centralized versus distributed queues

Task stealing with distributed queues
- Can compromise comm and locality, and increase synchronization
- Whom to steal from, how many tasks to steal, ...
- Termination detection
- Maximum imbalance related to size of task

(a) Centralized task queue
(b) Distributed task queues (one per process)
Determining Task Granularity

Task granularity: amount of work associated with a task

General rule:
- Coarse-grained => often less load balance
- Fine-grained => more overhead; often more communication and contention

Communication and contention actually affected by assignment, not size
- Overhead by size itself too, particularly with task queues

Reducing Serialization

Careful about assignment and orchestration (including scheduling)

Event synchronization
- Reduce use of conservative synchronization
  - e.g. point-to-point instead of barriers, or granularity of pt-to-pt
- But fine-grained synch more difficult to program, more synch ops.

Mutual exclusion
- Separate locks for separate data
  - e.g. locking records in a database: lock per process, record, or field
  - lock per task in task queue, not per queue
  - finer grain => less contention/serialization, more space, less reuse
- Smaller, less frequent critical sections
  - don't do reading/testing in critical section, only modification
  - e.g. searching for task to dequeue in task queue, building tree
- Stagger critical sections in time

Reducing Inherent Communication

Communication is expensive!

Measure: communication to computation ratio

Focus here on inherent communication
- Determined by assignment of tasks to processes
- Later see that actual communication can be greater

Assign tasks that access same data to same process

Solving communication and load balance NP-hard in general case
But simple heuristic solutions work well in practice
- Applications have structure!

Domain Decomposition

Works well for scientific, engineering, graphics, ... applications

Exploits local-biased nature of physical problems
- Information requirements often short-range
- Or long-range but fall off with distance

Simple example: nearest-neighbor grid computation

Perimeter to Area comm-to-comp ratio (area to volume in 3D)
- Depends on \( n, p \): decreases with \( n \), increases with \( p \)
Reducing Extra Work

Common sources of extra work:
- Computing a good partition
  - e.g., partitioning in Barnes-Hut or sparse matrix
- Using redundant computation to avoid communication
- Task, data and process management overhead
  - applications, languages, runtime systems, OS
- Imposing structure on communication
  - coalescing messages, allowing effective naming

Architectural Implications:
- Reduce need by making communication and orchestration efficient

\[
\text{Speedup} \leq \frac{\text{Sequential Work}}{\text{Max} (\text{Work} + \text{Synch Wait Time} + \text{Comm Cost} + \text{Extra Work})}
\]

Summary of Tradeoffs

Different goals often have conflicting demands
- Load Balance
  - fine-grain tasks
  - random or dynamic assignment
- Communication
  - usually coarse grain tasks
  - decompose to obtain locality: not random/dynamic
- Extra Work
  - coarse grain tasks
  - simple assignment
- Communication Cost:
  - big transfers: amortize overhead and latency
  - small transfers: reduce contention

Impact of Programming Model

Example: LocusRoute (standard cell router)

while (route_density_improvement > threshold)
    
    for (i = 1 to num_wires) do
        
        - rip old wire route out
        - explore new routes
        - place wire using best new route

    

Shared-Memory Implementation

Shared memory algorithm:
- Divide cost-array into regions (assign regions to PEs)
- Assign wires to PEs based on the region in which center lies
- Do load balancing using stealing when local queue empty

Good points:
- Good load balancing
- Mostly local accesses
- High cache-hit ratio
Message-Passing Implementations

Solution-1:
• Distribute wires and cost-array regions as in sh-mem implementation
• Big overhead when wire-path crosses to remote region
  - send computation to remote PE, or
  - send messages to access remote data

Solution-2:
• Wires distributed as in sh-mem implementation
• Each PE has copy of full cost array
  - one owned region, plus potentially stale copy of others
  - send frequent updates so that copies not too stale
• Consequences:
  - waste of memory in replication
  - stale data => poorer quality results or more iterations

=> In either case, lots of thinking needed on the programmer’s part

Case Studies

Simulating Ocean Currents
• Regular structure, scientific computing

Simulating the Evolution of Galaxies
• Irregular structure, scientific computing

Ray tracing
• Irregular structure, graphics

Steps in Creating a Parallel Program

4 steps: Decomposition, Assignment, Orchestration, Mapping
• Done by programmer or system software (compiler, runtime, ...)
• Issues are the same, so assume programmer does it all explicitly

Case 1: Simulating Ocean Currents

• Model as two-dimensional grids
• Discretize in space and time
  - finer spatial and temporal resolution => greater accuracy
• Many different computations per time step
  - set up and solve equations
• Concurrency across and within grid computations
### Time Step in Ocean Simulation

1. Put Laplacian of $\psi_1$ in $W_1$
2. Copy $W_1$ and $\gamma_1$ into $T_1$ and $W_2$
3. Put $W_2$ values in $W_3$
4. Initialize $\gamma_1$ and $\gamma_2$

- Add $f$ values to columns of $W_1$ and $W_3$
- Copy $W_3$ to $W_5$

- Put Jacobian of $(W_1, T_1)$ into $W_5$ and $W_3$
- Copy $T_1$ and $T_3$ into $\psi_1$ and $\psi_3$
- Copy $\psi_1M$ and $\psi_3M$ into $\psi_1$ and $\psi_3$
- Copy $T_1$, $T_3$ into $\psi_1M$ and $\psi_3M$

- Put Laplacian of $\psi_1M$, $\psi_3M$ in $W_7$
- Put Laplacian of $W_7$ in $W_4$
- Put Laplacian of $W_4$ in $W_7$

- Update the $\gamma$ expressions

- Solve the equation for $\nu_a$ and put the result in $\gamma_a$

- Compute the integral of $\psi_a$

- Compute $\psi - \nu_a = C(b) \psi_b$ (Note: $\psi_b$ and now $\psi$ are maintained in $\gamma_b$ matrix)$\quad$ Solve the equation for $\phi$ and put result in $\gamma_b$

- Use $\nu$ and $\phi$ to update $\psi_1$ and $\psi_3$

- Update streamfunction running sums and determine whether to end program

### Partitioning

**Exploit data parallelism**
- Function parallelism only to reduce synchronization

**Static partitioning within a grid computation**
- Block versus strip
  - Inherent communication versus spatial locality in communication
  - Load imbalance due to border elements and number of boundaries
- Solver has greater overheads than other computations

### Two Static Partitioning Schemes

- **Strip**
- **Block**

**Which approach is better?**

### Spatial Locality Example

- Repeated sweeps over 2-d grid, each time adding 1 to elements
- Natural 2-d versus higher-dimensional array representation

**Contiguity in memory layout**

- **Page straddles partition boundaries:** difficult to distribute memory well
- **Cache block straddles partition boundary**

- **Page does not straddle partition boundary**
- **Cache block is within a partition**

(a) Two-dimensional array

(b) Four-dimensional array
Tradeoffs with Inherent Communication

Partitioning grid solver: blocks versus rows
- Blocks still have a spatial locality problem on remote data
- Rowwise can perform better despite worse inherent c-to-c ratio
- Result depends on $n$ and $p$

Impact of Memory Locality
- Algorithmic = perfect memory system
- No Locality = dynamic assignment of columns to processors
- Locality = static subgrid assignment (infinite caches)

Execution Time Breakdown
- 1030 x 1030 grids with block partitioning on 32-processor Origin2000
- 4-d grids much better than 2-d, despite very large caches on machine
- Data distribution is much more crucial on machines with smaller caches
- Major bottleneck in this configuration is time waiting at barriers
- Imbalance in memory stall times as well
Case 2: Simulating Galaxy Evolution

- Simulate the interactions of many stars evolving over time
- Computing forces is expensive
- $O(n^2)$ brute force approach
- Hierarchical Methods take advantage of force law: $G \frac{m_1 m_2}{r^2}$

 muiti time-steps, plenty of concurrency across stars within one

Barnes-Hut

Locality Goal:
- particles close together in space should be on same processor

Difficulties:
- nonuniform, dynamically changing

Application Structure

- Main data structures: array of bodies, of cells, and of pointers to them
  - Each body/cell has several fields: mass, position, pointers to others
  - pointers are assigned to processes

Partitioning

Decomposition: bodies in most phases, cells in computing moments

Challenges for assignment:
- Nonuniform body distribution => work and comm. nonuniform
  - Cannot assign by inspection
- Distribution changes dynamically across time-steps
  - Cannot assign statically
- Information needs fall off with distance from body
  - Partitions should be spatially contiguous for locality
- Different phases have different work distributions across bodies
  - No single assignment ideal for all
  - Focus on force calculation phase
- Communication needs naturally fine-grained and irregular
Load Balancing

- Equal particles ≠ equal work.
  - Solution: Assign costs to particles based on the work they do

- Work unknown and changes with time-steps
  - Insight: System evolves slowly
  - Solution: Count work per particle, and use as cost for next time-step.

Powerful technique for evolving physical systems

A Partitioning Approach: ORB

Orthogonal Recursive Bisection:
- Recursively bisect space into subspaces with equal work
  - Work is associated with bodies, as before
  - Continue until one partition per processor
- High overhead for large number of processors

Another Approach: Costzones

Insight: Tree already contains an encoding of spatial locality.

- Costzones is low-overhead and very easy to program

Barnes-Hut Performance
- Speedups on simulated multiprocessor
- Extra work in ORB is the key difference
Execution Time Breakdown

- 512K bodies on 32-processor Origin2000
  - Static, quite randomized in space, assignment of bodies versus costzones

(a) Static assignment of bodies
(b) Semistatic costzone assignment

Problem with static case is communication/locality, not load balance!

Raytrace

Rays shot through pixels in image are called **primary rays**
  - Reflect and refract when they hit objects
  - Recursive process generates ray tree per primary ray

Hierarchical spatial data structure keeps track of primitives in scene
  - Nodes are space cells, leaves have linked list of primitives

Tradeoffs between execution time and image quality

Partitioning

*Scene-oriented approach*
  - Partition scene cells, process rays while they are in an assigned cell

*Ray-oriented approach*
  - Partition primary rays (pixels), access scene data as needed
    - Simpler; used here

Need dynamic assignment; use contiguous blocks to exploit spatial coherence among neighboring rays, plus tiles for task stealing

- A block, the unit of assignment
- A tile, the unit of decomposition and stealing

Could use 2-D interleaved (scatter) assignment of tiles instead

Other Techniques

Scatter Decomposition, e.g. initial partition in Raytrace

Preserve locality in task stealing
  - Steal large tasks for locality, steal from same queues, ...
Orchestration and Mapping

Spatial locality
- Proper data distribution for ray-oriented approach very difficult
- Dynamically changing, unpredictable access, fine-grained access
- Better spatial locality on image data than on scene data
  - Strip partition would do better, but less spatial coherence in scene access

Temporal locality
- Working sets much larger and more diffuse than Barnes-Hut
- But still a lot of reuse in modern second-level caches
  - SAS program does not replicate in main memory

Synchronization:
- One barrier at end, locks on task queues

Mapping: natural to 2-d mesh for image, but likely not important

Execution Time Breakdown

With task stealing
- Task stealing clearly very important for load balance

Implications for Programming Models

Shared address space and explicit message passing
- SAS may provide coherent replication or may not
- Focus primarily on former case

Assume distributed memory in all cases
Recall any model can be supported on any architecture
- Assume both are supported efficiently
- Assume communication in SAS is only through loads and stores
- Assume communication in SAS is at cache block granularity

Issues to Consider

Functional issues:
- Naming
- Replication and coherence
- Synchronization

Organizational issues:
- Granularity at which communication is performed

Performance issues
- Endpoint overhead of communication
  - (latency and bandwidth depend on network so considered similar)
- Ease of performance modeling

Cost Issues
- Hardware cost and design complexity
Naming

SAS: similar to uniprocessor; system does it all
MP: each process can only directly name the data in its address space
  • Need to specify from where to obtain or where to transfer nonlocal data
  • Easy for regular applications (e.g. Ocean)
  • Difficult for applications with irregular, time-varying data needs
    - Barnes-Hut: where the parts of the tree that I need? (change with time)
    - Raytrace: where are the parts of the scene that I need (unpredictable)
  • Solution methods exist
    - Barnes-Hut: Extra phase determines needs and transfers data before computation phase
    - Raytrace: scene-oriented rather than ray-oriented approach
    - both: emulate application-specific shared address space using hashing

Replication

Who manages it (i.e. who makes local copies of data)?
  • SAS: system, MP: program
Where in local memory hierarchy is replication first done?
  • SAS: cache (or memory too), MP: main memory
At what granularity is data allocated in replication store?
  • SAS: cache block, MP: program-determined
How are replicated data kept coherent?
  • SAS: system, MP: program
How is replacement of replicated data managed?
  • SAS: dynamically at fine spatial and temporal grain (every access)
  • MP: at phase boundaries, or emulate cache in main memory in software

Amount of Replication Needed

Mostly local data accessed => little replication
Cache-coherent SAS:
  • Cache holds active working set
    - replaces at fine temporal and spatial grain (so little fragmentation too)
  • Small enough working sets => need little or no replication in memory
Message Passing or SAS without hardware caching:
  • Replicate all data needed in a phase in main memory
    - replication overhead can be very large (Barnes-Hut, Raytrace)
    - limits scalability of problem size with no. of processors
  • Emulate cache in software to achieve fine-temporal-grain replacement
    - expensive to manage in software (hardware is better at this)
    - may have to be conservative in size of cache used
    - fine-grained message generated by misses expensive (in message passing)
    - programming cost for cache and coalescing messages
Communication Overhead and Granularity

Overhead directly related to hardware support provided

- Lower in SAS (order of magnitude or more)

Major tasks:

- Address translation and protection
  - SAS uses MMU
  - MP requires software protection, usually involving OS in some way
- Buffer management
  - fixed-size small messages in SAS easy to do in hardware
  - flexible-sized message in MP usually need software involvement
- Type checking and matching
  - MP does it in software: lots of possible message types due to flexibility
  - A lot of research in reducing these costs in MP, but still much larger

Naming, replication and overhead favor SAS

- Many irregular MP applications now emulate SAS/cache in software

Block Data Transfer

Fine-grained communication not most efficient for long messages

- Latency and overhead as well as traffic (headers for each cache line)

SAS: can using block data transfer

- Explicit in system we assume, but can be automated at page or object level in general (more later)
- Especially important to amortize overhead when it is high
  - latency can be hidden by other techniques too

Message passing:

- Overheads are larger, so block transfer more important
- But very natural to use since message are explicit and flexible
  - Inherent in model

Synchronization

SAS: Separate from communication (data transfer)

- Programmer must orchestrate separately

Message passing

- Mutual exclusion by fiat
- Event synchronization already in send–receive match in synchronous
  - need separate orchestration (using probes or flags) in asynchronous

Hardware Cost and Design Complexity

Higher in SAS, and especially cache-coherent SAS

But both are more complex issues

- Cost
  - must be compared with cost of replication in memory
  - depends on market factors, sales volume and other nontechnical issues
- Complexity
  - must be compared with complexity of writing high-performance programs
  - Reduced by increasing experience
**Summary for Programming Models**

Given tradeoffs, architect must address:
- Hardware support for SAS (transparent naming) worthwhile?
- Hardware support for replication and coherence worthwhile?
- Should explicit communication support also be provided in SAS?

**Current trend:**
- Tightly-coupled multiprocessors support for cache-coherent SAS in hw
- Other major platform is clusters of workstations or multiprocessors
  - currently don't support SAS in hardware, mostly use message passing

---

**Summary**

Crucial to understand characteristics of parallel programs
- Implications for a host or architectural issues at all levels

Architectural convergence has led to:
- Greater portability of programming models and software
  - Many performance issues similar across programming models too
- Clearer articulation of performance issues
  - Used to use PRAM model for algorithm design
  - Now models that incorporate communication cost (BSP, logP,...)
  - Emphasis in modeling shifted to end-points, where cost is greatest
  - But need techniques to model application behavior, not just machines